Comprehensive Resource Use Monitoring for HPC Systems with TACC Stats

Todd Evans*, William L. Barth*, James C. Browne†, Robert L. DeLeon‡,
Thomas R. Furlani‡, Steven M. Gallo‡, Matthew D. Jones‡, Abani K. Patra‡
*Texas Advanced Computing Center, Texas 78758, Email: {rtevans, bbarth}@tacc.utexas.edu
†The University of Texas at Austin, Texas 78712, Email: browne@cs.utexas.edu
‡University at Buffalo, SUNY, Buffalo, NY 14203, ‡Email: {rldeleon, furlani, smgallo, jonesm, abani}@buffalo.edu

***Abstract*—This paper reports on a comprehensive, fully automated resource use monitoring package, TACC Stats, which enables both consultants, users and other stakeholders in an HPC system to systematically and actively identify jobs/applications that could benefit from expert support and to aid in the diagnosis of software and hardware issues. TACC Stats continuously collects and analyzes resource usage data for every job run on a system and differs significantly from conventional profilers because it requires no action on the part of the user or consultants—it is always collecting data on every node for every job. TACC Stats is open source and downloadable, configurable and compatible with general Linux-based computing platforms, and extensible to new CPU architectures and hardware devices. It is meant to provide a comprehensive resource usage monitoring solution. In addition to describing TACC Stats, the paper illustrates its application to identifying production jobs which have inefficient resource use characteristics.**

I. Introduction

High performance computer architectures are complex and rapidly evolving. These systems are expensive, critical to advanced science and engineering and often heavily oversubscribed. There is thus an urgent need to utilize these systems as effectively as possible. Users seldom attempt comprehensive adaptation to new architectures or performance optimization. Prior to the package reported in this paper, there have been no self-contained, open source monitoring and analysis solutions which combine system-wide coverage, low overhead, a comprehensive monitoring scope, and that resolve resource use by job. The result is that neither users nor system managers/consultants have accurate knowledge concerning the effectiveness of these critical resources for science and engineering. This paper reports on a systematic and comprehensive approach to characterizing the resource use of jobs and applications on high performance computer systems, in particular, for identifying those jobs/applications which use system resources with significant inefficiency.

The package which implements this approach, TACC Stats, unifies and extends the measurements taken by Linux monitoring utilities such as systat/SAR, iostat [1], etc. and other Linux utilities. It resolves measurements by job and hardware device so that individual jobs/applications can be analyzed separately. TACC Stats collects data such as core-level CPU usage, socketlevel memory usage, swapping and paging statistics, system load and process statistics, system and block device counters, interprocess communications, filesystems usage (NFS, Lustre, Panasas), interconnect fabric traffic, and CPU counters and Uncore counters (e.g. counters from the Memory Controller, Cache and NUMA Coherence Agents, Power Control Unit). It can be readily modified or extended.

TACC Stats also provides a set of analysis and reporting tools which analyze TACC Stats resource use data and report jobs/applications with low resource use efficiency or that appeared to experience software or hardware issues. TACC Stats has extremely low overhead, allowing it to monitor every job and node and to be active at all times. It is initialized at the beginning of a job and collects data at specified intervals during job execution and once more at the end of a job. Each execution of TACC Stats takes less than 0.5s. When executed at the default interval (every 10 minutes) we estimate an overhead of less than 0.1%.

TACC Stats can be used to automatically generate analyses and reports such as average cycles per instruction (CPI), average and peak memory use, average and peak memory bandwidth use, interconnect traffic, and more on each job and over sets of jobs grouped according to user, application, project number, and date. These reports enable systematic identification of jobs, applications, or specific implementations of applications (such as building on one MPI stack vs another) which could benefit from architectural adaptation and performance tuning. In addition these analyses are potent tools for catching user mistakes such as allocating multiple nodes to a single-node shared-memory parallelized application or diagnosing system issues such as hardware and file-system failures.

TACC Stats has been collecting job-level performance data on all jobs run on the Lonestar and Stampede systems since September 2013. The current version of the performance monitoring framework has been implemented through modifications and enhancements to an older version of TACC Stats that was originally developed for and operated on the now retired Ranger system [2]. In this updated implementation, TACC Stats has been modified to readily support new CPU architectures and hardware devices. This update facilitated the current deployment of TACC Stats on the Lonestar and Stampede systems, and in particular supported the collection of data from numerous hardware counters available on Intel Sandy Bridge processors, which constitute much of Stampede's computational resources. The counters have since been extended to support Intel's Ivy Bridge processors. In addition, a suite of job-level metrics have been added which can be computed for any job by using new command line tools and an intuitive web interface, allowing the analysis of collected data by experts and non-experts alike. The tools and interface are currently in use by staff at TACC, and it is planned to make the web-based interface available to TACC's users.

While this data is used to inform diagnosis of system issues and decisions regarding hardware and software deployment, best practices for running applications, job scheduling, and allocation requests, we restrict the topics of this paper to the description of the TACC Stats package and its utility in the identification of applications that may be running unnecessarily inefficiently. We demonstrate the latter topic in part through several user application-based case studies.

TACC Stats is one component of an ongoing HPC systems analytics project, SUPReMM [3], which combines several efforts to produce an overall collection and analysis framework. Data collected through TACC Stats is fed into the eXtrem Digital Metrics on Demand framework, XDMoD [4], which was originally targeted at managers and providers of computational resources of the XSEDE [5] organization but is now also available as an open source product [6]. This framework originally provided for analysis of metrics including: number of jobs, CPUs consumed, wait time, and wall time, with minimum, maximum and the average of these metrics, in addition to many others. These metrics can be broken down by: field of science, institution, job size, job wall time, NSF directorate, NSF user status, parent science, person, principal investigator, and by resource. The SUPReMM project has enabled TACC Stats data to be included alongside these other metrics as a part of XDMoD. Additionally, TACC Stats leverages the Lariat [7] project at TACC to discover the usually unknown details about a job that includes executable names for parallel jobs; their working directories, size, creation date, and SHA1 hash; and shared libraries and environment modules that they may employ.

II. Related Work

A similar approach to system-wide, job-level monitoring was reported by Del Vento *et. al.* in [8] and was used for identifying poorly performing jobs and diagnosis of code failures. The infrastructure reported in [8] was, however, based on proprietary systems (IBM POWER/AIX) while TACC Stats is designed for open-source software based HPC clusters. In particular, the approach adopted by Del Vento *et. al.* relied on a command, `hpmstats`, that is available only on AIX systems to collect hardware counter data. A more actively developed measurement system, OVIS [9], utilizes a lightweight distributed metric service (LDMS) designed for Cray machines and also provides similar capabilities to TACC Stats. The open source system currently most comparable to TACC Stats in terms of functionality is the HOPSA project's LWM^2 performance screening tool [10]. LWM^2's reported data collection (particularly from hardware counters) is not nearly as comprehensive as TACC Stats though. Neither OVIS nor LWM^2 provide capabilities to resolve and analyze resource usage data by user, application, and/or project as TACC Stats/Lariat does.

There are many other open source and commercial resource usage monitoring tools which include: systat/SAR [1], iostat, CLUMON [11], PCP [12], Ganglia [13], and Nagios [14]. Only one of these, CLUMON, which uses data from PCP, is capable of resolving the data by job at the core level. CLUMON and PCP do not, however, collect data from hardware performance counters. Ganglia and Nagios do not resolve data by jobs or users. None of these systems is capable of resolving data at the application and/or project level.

While it may be possible to combine several of these tools to approximate the functionality of TACC Stats, such an approach would have significantly greater overhead and complexity than the single, comprehensive job-level resource usage monitor that TACC Stats is intended to be.

III. Description of TACC Stats

The currently downloadable version of the TACC Stats package [15] is composed of four modules:

1) `monitor` collects the resource usage data on every node
2) `pickler` processes the node-level data into job-level data
3) `analysis` provides data analysis and plotting tools
4) `site` provides a searchable and browsable web interface which dynamically generates plots and metrics along with providing access to raw data

TACC Stats is Python-based and can be configured by modifying a single file and installed using standard Python setup tools such as `pip` or `easy_install`. RPM's are also supported by the build system and are in fact how TACC Stats is deployed on TACC's systems.

The device data currently collected on Stampede are tabulated in Table I, where TACC Stats's abbreviated name for the device, full path to the source of the data, and a short description of the device are given. The exact address of all model-specific registers (MSR) and PCI configuration space counter controller registers and counter registers [16] are tabulated in the TACC Stats Doxygen documentation. The MSR registers and PCI configuration space (i.e. PCI device registers) are mapped to pseudo-files in Linux operating systems and control debugging, program execution tracing, and performance monitoring of CPUs and closely associated components. The explicit location of counters collected from PCI configuration space are given here because we found these to be challenging to locate. Detailed information about Intel Sandy Bridge core and uncore counters can be found in [16] and [17].

A. `monitor`

`monitor` is a 400KB-sized executable with several device-generic routines and device-specific modular routines for each device to be monitored. The collection of data from each hardware device is programmed in a templated manner—to add support for an entirely new hardware device a single C file with 4 routines must be written. The recommended method for adding hardware devices is included in the TACC Stats Doxygen documentation.

If data is not desired for a particular device that device can be simply disabled at compilation time. However, this is unnecessary because `monitor` checks for the presence of a particular device at run time and disables collection for devices which are not present. For example, `monitor` will check the

Device	Source	Description
block	/sys/block/DEV/stat	block device statistics (per device DEV)
cpu	/proc/stat	scheduler accounting (per CPU)
ib_ext	`infiniband/mad.h` routines	Infiniband Usage - Extended (per IB device)
intel_snb	/dev/cpu/CPU/msr	Sandy Bridge (per CPU = 0-15)
intel_snb_cbo	/dev/cpu/SOCKET/msr	Cache Agent (per socket SOCKET = [0,8])
intel_snb_hau	/proc/bus/pci/[7f,ff]/[0e.0,0e.1]	Home Agent (per socket)
intel_snb_imc	/proc/bus/pci/[7f,ff]/[10.0,10.1,10.4,10.5]	Memory Controller (per socket)
intel_snb_pcu	/dev/cpu/SOCKET/msr	Power Controller (per socket SOCKET = [0,8])
intel_snb_qpi	/proc/bus/pci/[7f,ff]/[08.2,09.2]	QuickPath Interconnect (per socket)
intel_snb_r2pci	/proc/bus/pci/[7f,ff]/13.1	Ring to PCI Express (per socket)
llite	/proc/fs/lustre/llite/stats	Lustre FS (per mount)
lnet	/proc/sys/lnet/stats	Lustre Network (per mount)
mdc	/proc/fs/lustre/mdc/MOUNT/stats	Metadata Client (per mount MOUNT)
mem	/sys/devices/system/node/DEV/meminfo	memory usage (per node)
net	/sys/class/net/DEV/statistics	network device usage (per device DEV)
nfs	/proc/self/mountstats	NFS file system usage (per device)
numa	/sys/devices/system/node/SOCKET/numastat	NUMA statistics (per socket SOCKET)
osc	/proc/fs/lustre/osc/MOUNT/stats	object storage client statistics (per mount MOUNT)
ps	/proc/stat	process statistics (per node)
sysv_shm	/proc/sysvipc/shm	SysV shared memory segment usage (per node)
tmpfs	proc/mounts	ram-backed file system usage (per node)
vfs	/proc/sys/fs/dentry-state	dentry/file/inode/cache usage (per node)
vmstat	/proc/vmstat	virtual memory statistics (per node)

TABLE I. LIST OF DEVICE DATA COLLECTED ON STAMPEDE WITH DATA SOURCE AND DESCRIPTION.

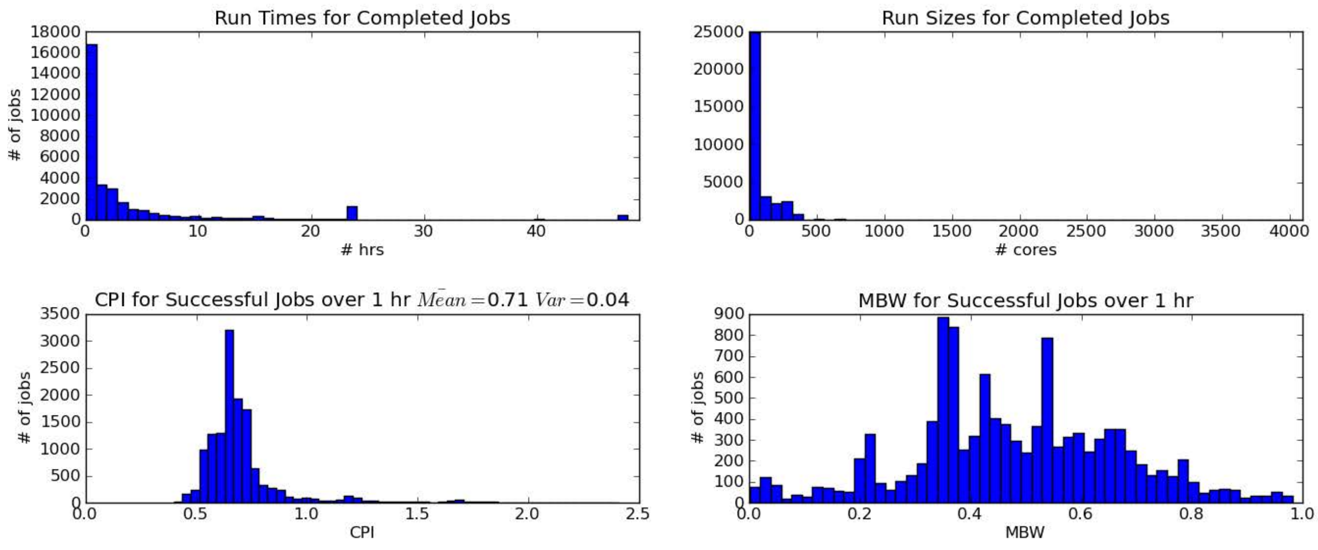

Fig. 1. Histograms generated from executables run (since September 2013) with a name that contains the substring `vasp_std` (Vienna Ab initio Simulation Package [18]). The plots show histograms for the number of jobs with a particular (from left to right, top to bottom): run time, size in cores, average CPI, and memory bandwidth as a percentage of peak bandwidth.

`cpuid` vendor and signature before programming or reading data from the `msr` file. If the CPU signature doesn't match the expected one or the device doesn't exist `monitor` will ignore that device. This allows the same build of `monitor` to run on nodes with disparate architectures and still correctly collect the data from each node. In fact, all devices can be enabled at all times, whether they are available on a particular platform or not, and `monitor` will correctly run, ignoring any missing devices.

In `monitor`'s current configuration on Lonestar and Stampede, it is run in the prolog of the job scheduler, programming the counters where necessary and collecting the initial values from the counters. The initial run takes ∼1s. `monitor` is then run every ten minutes using the standard UNIX task scheduling facility, `cron`. Each of these collections requires ∼0.5s. It is then called at the end of the job in the epilog, again requiring ∼0.5s. This results in at least 2 samples per job.

During each execution of `monitor`, the data is written to a text file local to the compute node on which the module is running. The text file need not be local; however, we have not tested the load induced by writing to a distributed filesystem, e.g. Lustre. Each entry in the text file is labeled by Job ID and POSIX time. Every 24 hours a new file is started and the old file is moved from each compute node to a centralized location.

B. `pickler`

Every 24 hours the `pickler` module processes the node-level data collected by `monitor` into job-level data and stores each job's data in a Python pickle file. "Pickling" is an in-built Python language process for serializing Python objects into byte streams which can be written to storage or sent across the network. These byte streams can be decoded and deserialized back into native Python objects [19]. The Python pickle file is composed of nested Python dictionaries indicating Job ID, nodes, device type, device name or number if multiple devices of a particular type are present, events, and time stamps for each data sample. The data is regularized as it is picked—individual nodes in a jobs may run their cron tasks at slightly different times due to the overall system load on each node resulting in slightly different POSIX time stamps that must be treated. For most jobs processed this discrepancy in times is non-existent or very modest. If discrepancies are found `pickler` will find the node with the median number of time records and use that node's time stamps as the standard. Other nodes' records are then matched to those time stamps, replacing their time stamps with the standard time stamp closest in value. We are currently investigating replacing this procedure with an interpolation scheme.

In addition, data anomalies due to counter overflow and wrapping, although occurring only rarely, are corrected. We have calculated that for the counters currently supported, overflow cannot occur more than once in the 10 minute interval between collections. The current infrastructure cannot account for multiple overflows in a collection interval, so care should be taken by `monitor` users who are interested in counters other than those provided in TACC Stats. Roughly 2.5GB of pickled data from Stampede and 300MB from Lonestar are generated every day.

C. `analysis`

This module contains a suite of tests and plotting routines that can be run on arbitrary sets of jobs. Every test can be invoked through a single command-line tool, `job_sweeper.py`, with arguments controlling the range of jobs to test, thresholds to compare metrics against, and the number of shared-memory parallel processes to use. Currently we run the following tests every 24 hours:

- Imbalance: The ratio of standard deviation over the mean of an event is computed over all cores at each time point. This ratio is averaged over time and compared to a threshold of 1.0. The default event is the number of times a cache line is loaded into the L1 data cache, labeled `LOAD_L1D_ALL`.
- High CPI: We compute the average Cycles Per Instruction over all cores and compare the result to a threshold of 1.0. The average CPI of jobs on Stampede over the past three months is roughly 0.6.
- Idle: This test uses several counters to determine if a particular node is idle for most of a job's runtime. A representative plot of such a job is shown in Figure 2.
- Catastrophe: This test finds step-function like behavior in certain events, by default `LOAD_L1D_ALL`, on a node with a running job. For an example plot of such a job see Figure 3. The catastrophic crash shown in this job was due to the failure of a node. This test can also be used to test for compilation activity on compute nodes (often considered a waste of system resources), where low resource usage for most devices is followed suddenly by high activity. An example of such a job is shown in Fig. 4

In the above analyses, it is usually critical to know how many cores on each node the user has intended to use. For example, if a user intentionally idles half the cores on each node in order to double the amount of memory available to each core that they do use, we should account for this when trying to detect an catastrophic drop in performance or a high CPI. We derive this information from our Lariat data where possible [7], and ignore jobs for which we cannot determine how many cores were supposed to be in use.

Figures 2 and 3 were generated using the `analysis` module command line tool `job_plotter.py`. Similar figures for any combination of devices can be generated by using the appropriate command line arguments.

D. `site`

The `site` module provides a web site interface to the TACC Stats package. It builds a database of the job metadata and metrics through daily updates and serves the job time series data as media files. Currently, five metrics are computed and stored in the database for every successful production level job (for jobs over 1 hour): mean cycles per instruction, mean memory bandwidth, idleness, step function similarity, and the memory usage high water mark.

A representative screenshot of all jobs run since September 2013 which contain `vasp_std` in the executable name (jobs

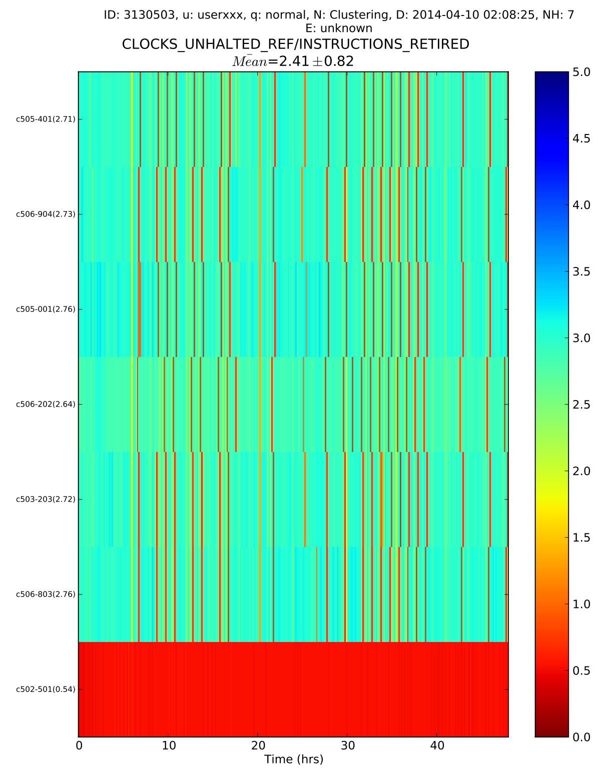

Fig. 2. A job flagged for running with idle nodes. The nodes are on the y-axis and the time stamp is on the x-axis. In this figure we are showing the number of Reference Clock Cycles per Instruction Retired. The core- and time-averaged CPI per node can be seen by each node name on the y-axis and the average over all nodes in the plot title. It can be inferred from this plot that only one of seven nodes was active for this job.

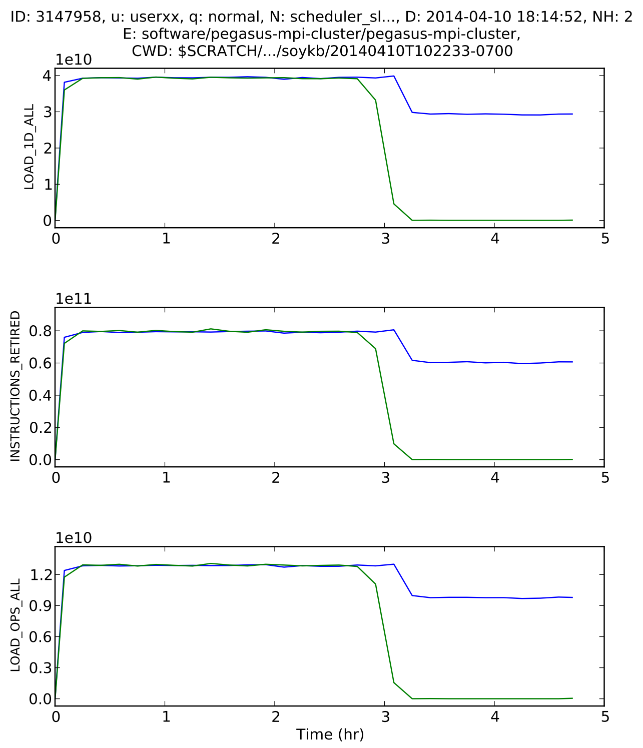

Fig. 3. Multiple CPU counters are seen to crash simultaneously on one of the nodes at around hour 3 for this job. This job was flagged by the Catastrophe test.

using TACC's standard build of the Vienna Ab initio Simulation Package [18]) are shown in Figure 1. The histograms from left to right, top to bottom show: run times, number of cores used, CPI with mean and variance computed, and memory bandwidth as a percentage of peak bandwidth. Jobs are currently searchable by date, user, Project, executable, and Job ID. These four histograms are generated automatically using the results of each search. Performance metrics such as CPI or Memory Bandwidth are particularly useful because abnormal jobs, jobs which had significantly higher or lower CPI than the average for a particular application, can be flagged and investigated further.

Each job entry on the web interface has the Job ID, User ID, user name, project, executable, start time, end time, run time, queue, job name, exit status, cores and nodes used, and CPI if computed. Every job's resource usage can be examined in detail using the plots automatically generated in the web interface such as those seen in Figures 2 and 3, and from the raw data. A particular combination of plots we've found to be useful for characterizing a job's performance is shown in Figure 4. This plot is generated and displayed every time a particular job is selected in the browser. From top to bottom, the plots in Figure 4 show for each node the number of vectorized instructions issued, memory bandwidth, memory usage, Lustre network bandwidth, Infiniband bandwidth (less Lustre traffic), and CPU user fraction.

IV. Case Studies

We use five case studies to demonstrate the utility of the TACC Stats package. Each of these case studies was the result of a particular job being flagged by at least one of our nightly tests. We present the first case study in the greatest detail and summarize the results of the other four. The general procedure used in these case studies is the following:

1) A job is flagged and we contact the user and verify that the user believes they ran the application correctly for the job in question. This is an important step because users regularly run applications in a inefficient manner intentionally for a variety of reasons.
2) We request from the user information regarding the location, configuration, and compilation of the application along with batch submission scripts and permission to work with their files.
3) This information is then used to profile the code using a standard profiler or if profiling seems unnecessary we make a recommendation to the user.

Any profiling performed in the following used the Intel VTune profiler [20] in hotspot mode. VTune can generate an individual performance report for each MPI task and indicates the amount of time each task spent on a line of code. For all case studies presented here the behavior of all MPI tasks were similar. The names of user applications have been altered to maintain anonymity except in the case of community software.

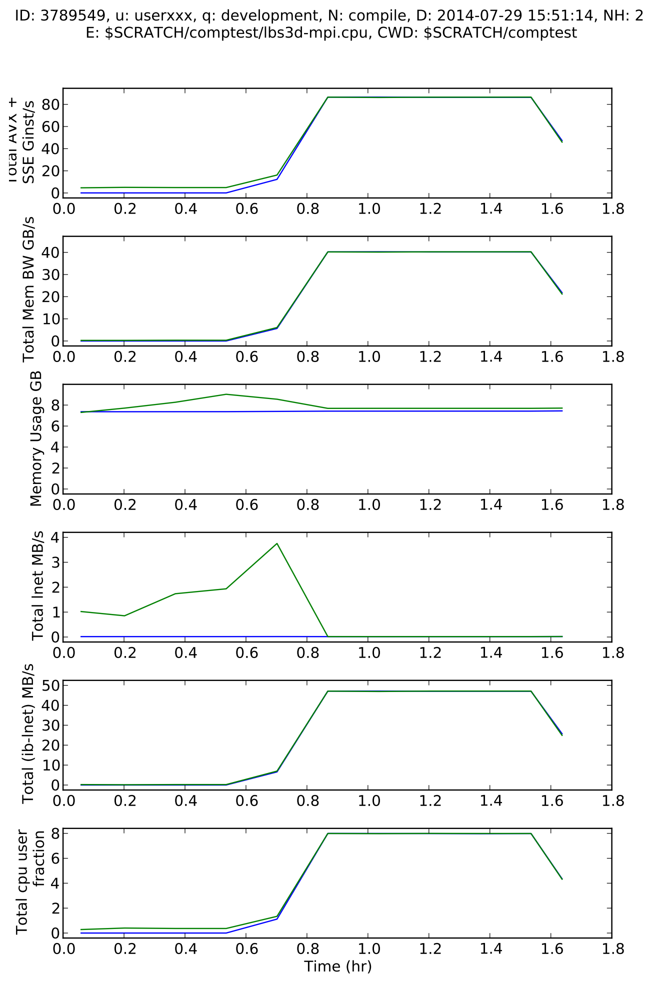

Fig. 4. Multiple CPU counters are seen to start simultaneously after a long idle period. This signature is indicative of a compile and execute (seen here to start around 45 mn) workflow which is discouraged on TACC compute nodes. The Lustre traffic (the plot labeled by **lnet**) during compilation arises from accesses to the user's source code on the distributed filesystem. This job was flagged by the Catastrophe test. We find this particular combination of plots to be particularly useful for characterizing a job's performance (from top to bottom): vectorized instructions issued, memory bandwidth, memory usage, Lustre network bandwidth, Infiniband bandwidth (less Lustre traffic), and CPU user fraction. Each line represents a distinct node.

A. *Case 1:* `Turbulence`

The `Turbulence` application is a 3D Navier-Stokes solver implemented in Fortran and capable of hybrid distributed- and shared-memory parallelization (MPI/OpenMP). The solver decomposes the simulated system's geometry over MPI tasks and parallelizes computations within tasks using OpenMP. The application instance we considered, *bcRoughnessCylinder*, was identified by TACC Stats as potentially underperforming according to the cycles per instruction (CPI) metric with a value of 1.1 (compare to the average CPI = 0.6). The geometry was not easily modifiable so we ran the solver on 32 nodes.

After profiling the code we found that generally the solver was written in an efficient manner, resulting in few obvious improvements to be made. The most significant hotspots in the solver could be attributed to non-locality of data, both from distributed- and shared-memory. For our profiling we ran the solver for 25 time steps for a total run time of 1350s. The main loop required 983s of this time (the user typically runs the solver for hundreds of time steps but believed our shorter runs would be sufficient for profiling the main loop). The top hotspots for an example MPI task are shown in Table II (note that the time spent in a function is aggregated over all threads calling the function).

Function	Time (s)	File
`passmpidouble`	976	`parallel.f90`
`u2v`	781	`mainvars.f90`
`comppointfluxzeta`	706	`termsInviscid.f90`

TABLE II. TOP HOTSPOTS FOR UNOPTIMIZED `TURBULENCE` BINARY.

The solver spent the most time in the user defined `passmpidouble` function. `passmpidouble` updates components of a large array (`Utemp`) shared between MPI tasks. We made no attempt to optimize this function. We did find the parallelization scheme of 2 MPI tasks per node and 8 OpenMP threads per task to result in the shortest run time. Other parallelization schemes were tried resulting in worse performance. This result was confirmed and expected by the user.

The `u2v` function consumed the second largest proportion of solver time. This function performs intrinsic Fortran operations on arrays to convert the application's conservative variable arrays (`Utemp`) to primitive variable arrays (`primVar`). The OpenMP directive `WORKSHARE` was used to parallelize this section of code over all OpenMP threads. We observed in the VTune report that a significant portion of time spent in this function was at the `END WORKSHARE` directive, indicating overhead due to thread imbalance. This is likely because the `WORKSHARE` directive requires synchronization at each line within its parallelized block and may not be effectively distributing the workload among threads, some of which suffer more greatly from memory non-locality. We optimized `u2v` by converting the `WORKSHARE` block composed of Fortran array intrinsics to a loop explicitly over array elements parallelized with the OpenMP parallel do-loop pragma: `DO SCHEDULE (GUIDED)`. This allows the OpenMP runtime system to schedule threads more efficiently.

The `comppointfluxzeta` function computes a number of quantities using the primitive and conservative arrays. Most of the time spent in this function can be attributed to computations using data in shared-memory with poor spatial locality. This is most likely because the primitive and temporary variables are 4D arrays and large in size. The code developers minimized memory access overheads by first copying the 4D arrays to temporary arrays of smaller dimensionality before performing several computations involving these arrays. We observed in `comppointfluxzeta` that the temporary arrays with reduced dimensionality were not being used in several places that they could be. This was changed in the optimized version resulting in a 20% reduction in time spent in the function. These results are summarized in Table III. Note that the timings shown for the `u2v` and `comppointfluxzeta` routines are aggregated over the 8 threads, while `Main Loop` is just the overall run time.

Function	Pre(s)	Post(s)	Pre/Post(%)
`Main Loop`	983	886	90
`u2v`	781	185	24
`comppointfluxzeta`	706	561	79

TABLE III. TIMINGS PRE- AND POST-OPTIMIZATION.

We reduced the overall runtime of the main loop by 10% with the optimizations to the `u2v` and `comppointfluxzeta` functions. This improvement is not dependent on which MPI task is analyzed. We identified additional potential improvements but believed these would require significant changes throughout the application's 70,000 lines of code.

B. Case 2: `Singularity`

The `Singularity` application is a distributed-memory parallelized General Relativity code that aims to simulate binary black hole systems. It is currently developed and used by a single Stampede user. The executable consists of a C binary which performs the computations and also runs Perl scripts and makes system calls. This application was our 3rd highest service unit (SU) consumer for January 1-March 31, 2014 that was flagged for a high CPI, with an SU usage of 239,631 and typical CPI of $\sim$1.15.

An inspection of the code revealed that it was compiled with no optimization flags and contained numerous extraneous I/O intensive calls to the UNIX `cat` and `rm` commands. These calls were being issued from a single MPI process out of 512 for the sole purpose of concatenating all the files written from each process into a single file, and then removing the 512 files. This requires an additional read and write to the Lustre storage servers and unnecessary operations on the Lustre metadata server. It is also work that can be done outside the job environment after it has completed. The user stated this workflow was necessary on a previous computing platform and assumed it was on Stampede as well.

We enabled `icc`'s optimization level `O3`, enabled the `xhost` SIMD vectorization flag, and removed the extraneous system calls. Our user reported a 26% decrease in runtime for production runs of 11,000 iterations from 19 to 14 hours.

C. Case 3: MILC `su3_mode` and `su3_hmc_APBC`

The MILC collaboration has a freely available software suite known as the MILC code [21][22]. This code simulates Quantum Chromodynamics (QCD) through a computationally intensive hybrid of Monte-Carlo methods and Molecular Dynamics known as Lattice Field Theory. A specific MILC application, `su3_mode`, was flagged with a CPI of 1.2. This should be compared with the average CPI over all MILC applications' of 0.72.

After profiling we saw that the code spent the most time in MPI calls. We did not attempt to optimize the use of the MPI library. Of routines within the application source code, we found the job spent the most time in the matrix vector multiplication routines `mult_su3_mat_vec_sum_4dir` and `mult_adj_su3_mat_vec_4dir`. We discovered that there were SSE2 inline assembly macros for these functions in a newer version of MILC which these particular MILC users at TACC were unaware of. After enabling these macros in the user's code we found a 30% reduction in time spent in these routines. The reduction in run time due to the modifications was, however, hard to quantify due to the stochastic nature of the underlying algorithm which leads to inconsistent run times from instance to instance.

The user then directed us to another application from MILC which their collaboration regularly runs on Stampede with the rationale that the application, `su3_hmc_APBC`, shares many of the same routines with `su3_mode`. We profiled this application and observed the hotspots to be dominated by 3 different 3×3 complex matrix multiplication routines. We used the SSE2 inline assembly routines for these routines as well, and found a reliable 11% decrease in run time for the entire application. Similar improvement was subsequently seen by the user in several other MILC applications that rely on these routines. The user base of MILC on TACC's systems has been notified of the benefits to be gained from using these optimized routines.

D. Cases 4 & 5: `Genome_map` & `SI_Screening`

Jobs run using these application were flagged while testing for idle nodes. The `Genome_map` user requested two nodes, however, their application was only capable of shared memory parallelization and was actually running on just one node. The `SI_Screening` user was requesting seven nodes but only running the application on one (Fig. 2). The users have been notified and they have since stopped submitting multi-node jobs for these applications. These egregious wastes of system and user resources would have gone unnoticed without the TACC Stats package. Perhaps surprisingly, several such jobs consuming significant SUs are identified daily on TACC's Stampede system.

V. FUTURE WORK

A. Validation

We currently only use a small percentage of the data available to us for our analyses. There are several reasons for this:

- Not all of the data is obviously useful.
- The data is incompletely understood. We are still interpreting many of the Intel counters and some of the counters from the Linux OS. The documentation regarding performance counters is often ambiguous or incomplete.
- The data is imperfect. For example, the vector floating-point instruction counters on Stampede's Intel Sandy Bridge cores count instructions issued not instructions retired. These instructions may be reissued many times within the pipeline while waiting for operands to be fetched from higher cache levels or main memory. As a result, we have seen order-of-magnitude higher differences in expected and measured counts for floating-point operations. We do, however, still use this data as a rough guide as to whether the application was

vectorized during compilation and was performing work while executing.

- We will continue to extend the range of hardware devices supported. In the near term, we hope to develop support for the Intel Xeon Phi Co-processor.

The second and third issues are closely related and could be grouped under the term validation. Validation is a labor intensive and on-going effort for the project. There are also degrees of validation and data must be interpreted with this in mind. For example, it is often possible to perform consistency checks between counters that should be related in a predictable way, but not the raw value of the counter due to indeterminism. In each case, the counters used here have either been validated with test codes or their deficiencies understood.

B. Package Distribution

The TACC Stats package is configurable and should be compatible with all Linux-based HPC platforms. However, we have currently only been able to test this compatibility on TACC's Lonestar and Stampede systems. These systems both use CentOS [23] as their operating systems but have compute nodes with distinct chip architectures. It remains to be seen how robust the configuration process is on other Linux distributions and how straightforward adding additional hardware support is for non-developers. We plan to work closely with any initial adopters in the HPC community to resolve unforeseen deficiencies in compatibility and provide the appropriate code for architectures or devices that are not currently available in TACC Stats. Eventually, we hope to develop a general procedure for the inclusion of code that provides new hardware support into the TACC Stats repository so that anyone may contribute.

The command line tools and web interface are currently used by consultant staff at TACC. In the near term future we expect to fully integrate this data into the XDMoD Metrics on Demand [3][4] infrastructure so that users can browse their own jobs through this interface and examine the associated plots and analyses relevant to performance.

We are also currently working with Louisiana State University, NCAR and Purdue to install TACC Stats on HPC systems at those institutions. This will aid in developing portability and robustness across platforms.

VI. Summary

The TACC Stats package provides a job-level, system-wide resource usage monitoring solution for HPC systems. It continuously monitors all nodes and all jobs. The jobs' resource usage data are automatically consolidated and tested for inefficiencies and software and hardware failures. This data is searchable and browsable through the web interface, with plots and tests generated automatically via the web interface. In addition, command line tools are available for customizable tests and/or plots. TACC Stats is open source, downloadable, and configurable.

We included the Case Studies to provide examples of how we are currently using TACC Stats. We are using TACC Stats as a force multiplier—a single consultant can detect one inefficient application or job among thousands and then take a closer look at that application or job, removing any obvious inefficiencies. This will result in large performance gains in some instances and smaller gains in others as demonstrated in our 5 case studies. While the performance improvements made in this paper were for the most part modest, we note that the flagged jobs were run in production mode. Applications run in production mode on TACC's systems, in our experience, rarely have very large inefficiencies that are straightforward to correct except in the case of gross user error such as completely neglecting optimization flags or running shared-memory code in a distributed memory fashion (as seen in Case Studies 4 and 5). Intensive expert consultation probably would have achieved larger gains in performance than seen in this paper. While we did not have a consultant to devote to every flagged application, a single consultant was able to identify the applications for the Case Studies and apply obvious optimizations to each case study and gain a significant performance improvement.

VII. Acknowledgements

References

[1] "SYSSTAT." [Online]. Available: http://sebastien.godard.pagesperso-orange.fr

[2] J. Hammond, "Tacc_stats: I/O performance monitoring for the instransigent," in *Invited Keynote for the 3rd IASDS Workshop, 2011*, 2011, pp. 1–29.

[3] J. C. Browne, R. L. DeLeon, C.-D. Lu, M. D. Jones, S. M. Gallo, A. Ghadersohi, A. K. Patra, W. L. Barth, J. Hammond, T. R. Furlani, and R. T. McLay, "Enabling comprehensive data-driven system management for large computational facilities," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 86:1–86:11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503230

[4] T. R. Furlani, M. D. Jones, S. M. Gallo, A. E. Bruno, C.-D. Lu, A. Ghadersohi, R. J. Gentner, A. Patra, R. L. DeLeon, G. von Laszewski, F. Wang, and A. Zimmerman, "Performance metrics and auditing framework using application kernels for high-performance computer systems," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 7, pp. 918–931, 2013. [Online]. Available: http://dx.doi.org/10.1002/cpe.2871

[5] "XSEDE Home." [Online]. Available: https://www.xsede.org

[6] "Open XDMoD." [Online]. Available: http://xdmod.sourceforge.net

[7] "Lariat." [Online]. Available: https://github.com/TACC/Lariat

[8] D. Del Vento, T. Engel, S. Ghosh, D. Hart, R. Kelly, S. Liu, and R. Valent, "System-level monitoring of floating-point performance to improve effective system utilization," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, Nov 2011, pp. 1–6.

[9] J. Brandt, B. Debusschere, A. Gentile, J. Mayo, P. Pebay, D. Thompson, and Wong, "Ovis-2: A robust distributed architecture for scalable ras. in 22nd ieee international parallel and distributed processing symposium," in *4th Workshop of System Management Techniques, Processes, and Services*, 2008.

[10] "HOPSA-Holistic Performance System Analysis." [Online]. Available: http://www.vi-hps.org/projects/hopsa/overview

[11] "CLUMON." [Online]. Available: http://clumon.ncsa.illinois.edu

[12] "PCP." [Online]. Available: http://oss.sgi.com/projects/pcp

[13] "GANGLIA." [Online]. Available: http://ganglia.sourceforge.net

[14] "NAGIOS." [Online]. Available: http://www.nagios.org

[15] "TACC Stats." [Online]. Available: https://github.com/TACCProjects/tacc_stats

[16] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel, February 2014.

[17] *Intel® Xeon® Processor E5-2600 Product Family Uncore Performance Monitoring Guide*, Intel, March 2012.

[18] “The VASP Site.” [Online]. Available: https://www.vasp.at

[19] “Python object serialization.” [Online]. Available: https://docs.python.org/2.7/library/pickle.html

[20] J. Reinders, *VTune Performance Analyzer Essentials: Measurement and Tuning Techniques for Software Developers. Engineer to Engineer Series*. Intel Press, 2005.

[21] C. Bernard, M. C. Ogilvie, T. A. Degrand, C. E. Detar, S. A. Gottlieb, A. Krasnitz, R. L. Sugar, and D. Toussaint, “Studying quarks and gluons on MIMD parallel computers,” *International Journal of High Performance Computing Applications*, vol. 5, pp. 61–70, 1991.

[22] MIMD Lattice Computation (MILC) Collaboration. [Online]. Available: http://physics.indiana.edu/~sg/milc.html

[23] “CentOS Project.” [Online]. Available: www.centos.org